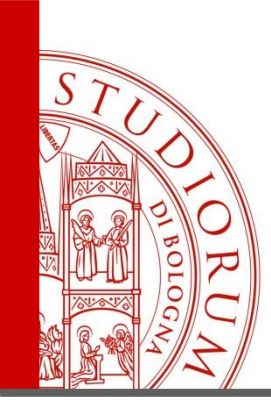
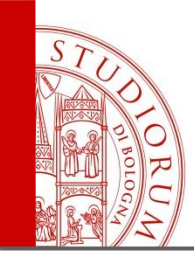


Università degli Studi di Bologna
Corso di Laurea in Ingegneria Informatica

Delegati ed Eventi

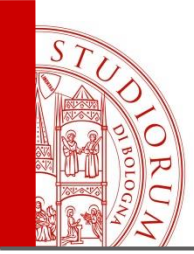
Ingegneria del Software T

Prof. MARCO PATELLA
Dipartimento di Informatica – Scienza e Ingegneria (DISI)



Delegati

- Sono oggetti che possono contenere **il riferimento** (*type safe*) **a un metodo**, tramite il quale il metodo può essere invocato
- **Oggetti funzione** (*functor*)
  oggetti che si comportano come una funzione (metodo)
- Simili ai puntatori a funzione del C/C++,
  ma *object-oriented* e molto più potenti
- Utilizzo standard: funzionalità di ***callback***
  - **Elaborazione asincrona**
  - **Elaborazione cooperativa** (il chiamato fornisce una parte del servizio, il chiamante fornisce la parte rimanente – es. qsort in C)
  - **Gestione degli eventi** (chi è interessato a un certo evento si registra presso il generatore dell'evento, specificando il metodo che gestirà l'evento)

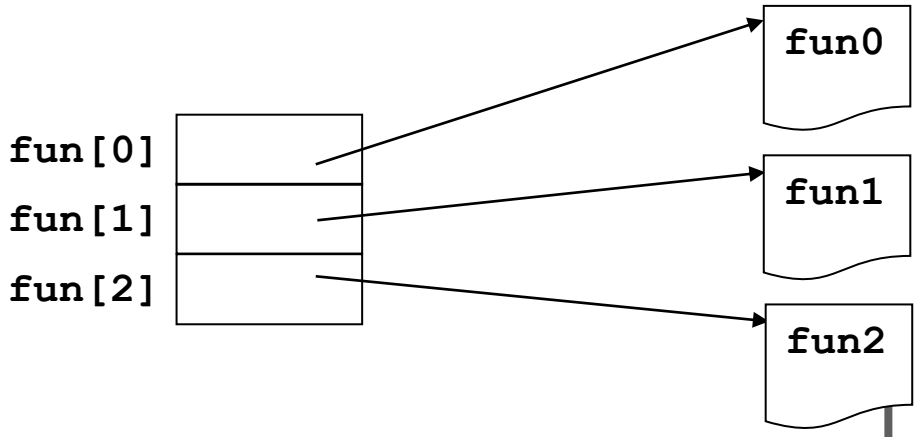


C/C++: Puntatori a funzioni

```
int funX(char c);
int funY(char c);
int (*g)(char c) = NULL;
...
g = cond1 ? funX : funY;
oppure:  g = cond1 ? &funX : &funY;
...
… g('H') …  ≡  … (*g)('H') …
```

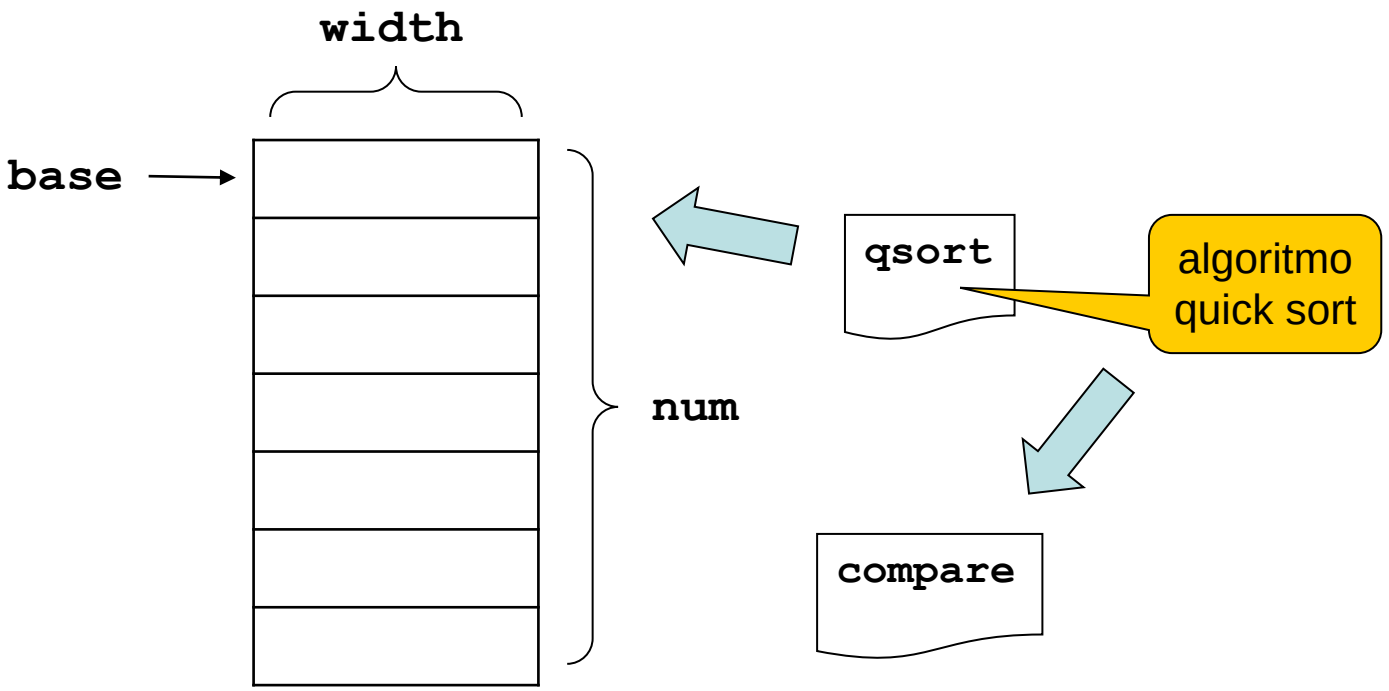
C/C++: Array di puntatori a funzioni

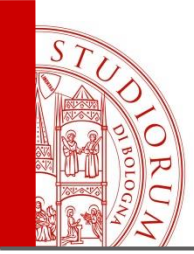
```
void fun0(char *s);
void fun1(char *s);
void fun2(char *s);
void (*fun[])(char *s) =
   { fun0, fun1, fun2 };
...
fun[m]("stringa di caratteri");  ≡
   (*fun[m])("stringa di caratteri");
```



C/C++: Elaborazione cooperativa

```
void qsort(void *base, int num, int width,
    int (*compare)(void *, void *));
```





Delegati

- **Dichiarazione** di un nuovo **tipo** di **delegato**
  che può contenere il riferimento a un metodo
  che ha un unico argomento intero e restituisce un intero:

```
delegate int Azione(int param);
```

- **Definizione** di un **delegato**:

```
Azione azione;
```

- **Inizializzazione** di un delegato:

```
azione = new Azione(nomeMetodoStatico);
azione = new Azione(obj.nomeMetodo);
azione = nomeMetodoStatico; // C# 2.0
azione = obj.nomeMetodo;    // C# 2.0
```

- **Invocazione del metodo** referenziato dal delegato:

```
int k1 = azione(10);
```

Esempio 3.1

Delegati: *Multicasting*

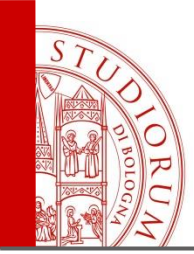
- È possibile assegnare al delegato una **lista di metodi** (**invocation list**)
- All'atto della chiamata del delegato, i metodi vengono invocati
  - **in sequenza**
  - **in modo sincrono**
- Per **aggiungere un metodo** alla lista: **+=**

```
Azione azione = Fun1;
… azione(10) …  // Fun1(10)
azione += Fun2;
… azione(10) …  // Fun1(10), Fun2(10)
```

- Per **togliere un metodo** dalla lista: **-=**

```
azione -= Fun1;
… azione(10) …  // Fun2(10)
```

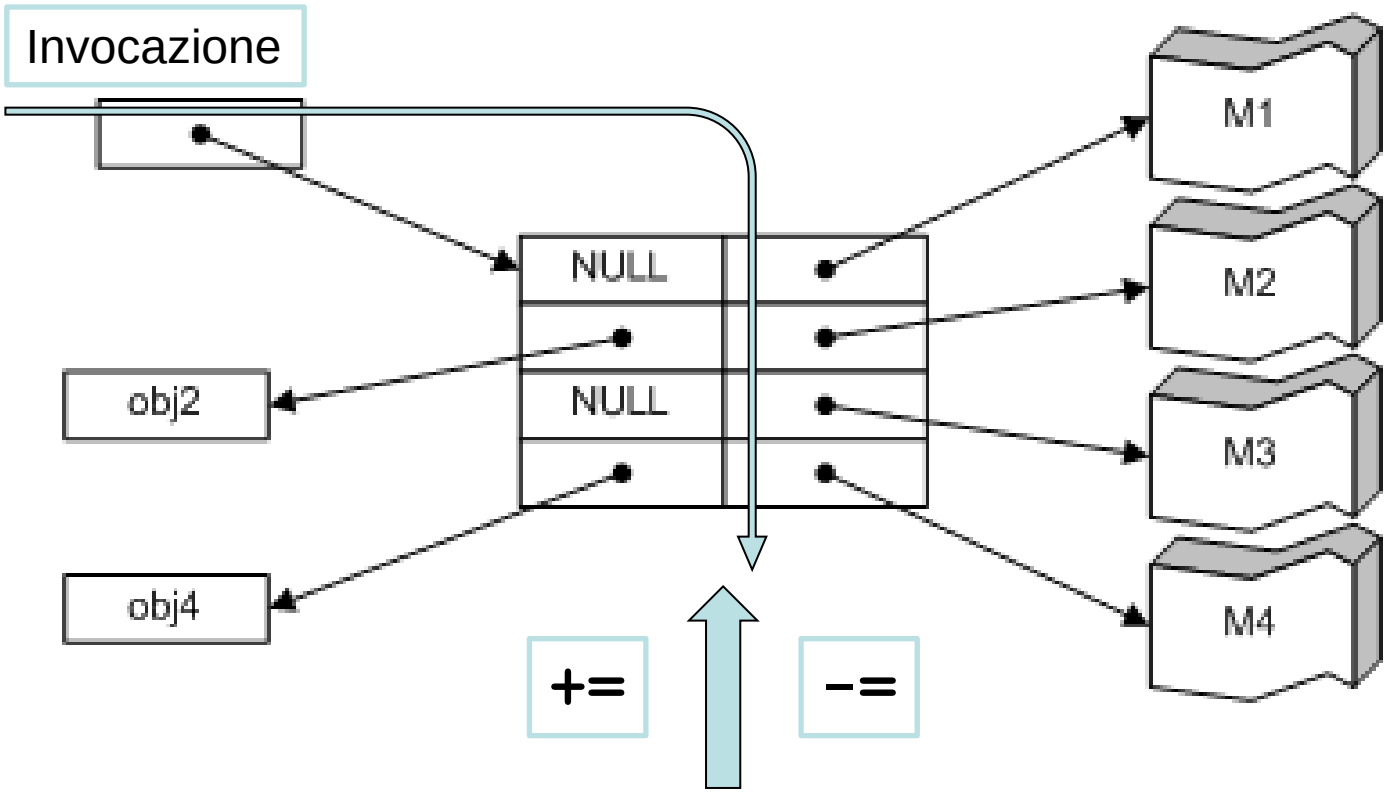
Esempio 3.2



Delegati

- Un’istanza di delegato incapsula uno o più metodi (con una lista di parametri e tipo restituito specifici), ciascuno dei quali è indicato come entità invocabile (**callable entity**)
  - per i **metodi statici**, un’entità invocabile consiste solo in un metodo
  - per i **metodi di istanza**, un’entità invocabile consiste in un'istanza e un metodo su quell'istanza
- Un delegato
  - applica solo una singola **firma del metodo** (non un nome)
  - non conosce o non si preoccupa della classe dell'oggetto a cui fa riferimento
- Ciò rende i delegati utili per chiamate anonime (**anonymous invocation**)

Delegati

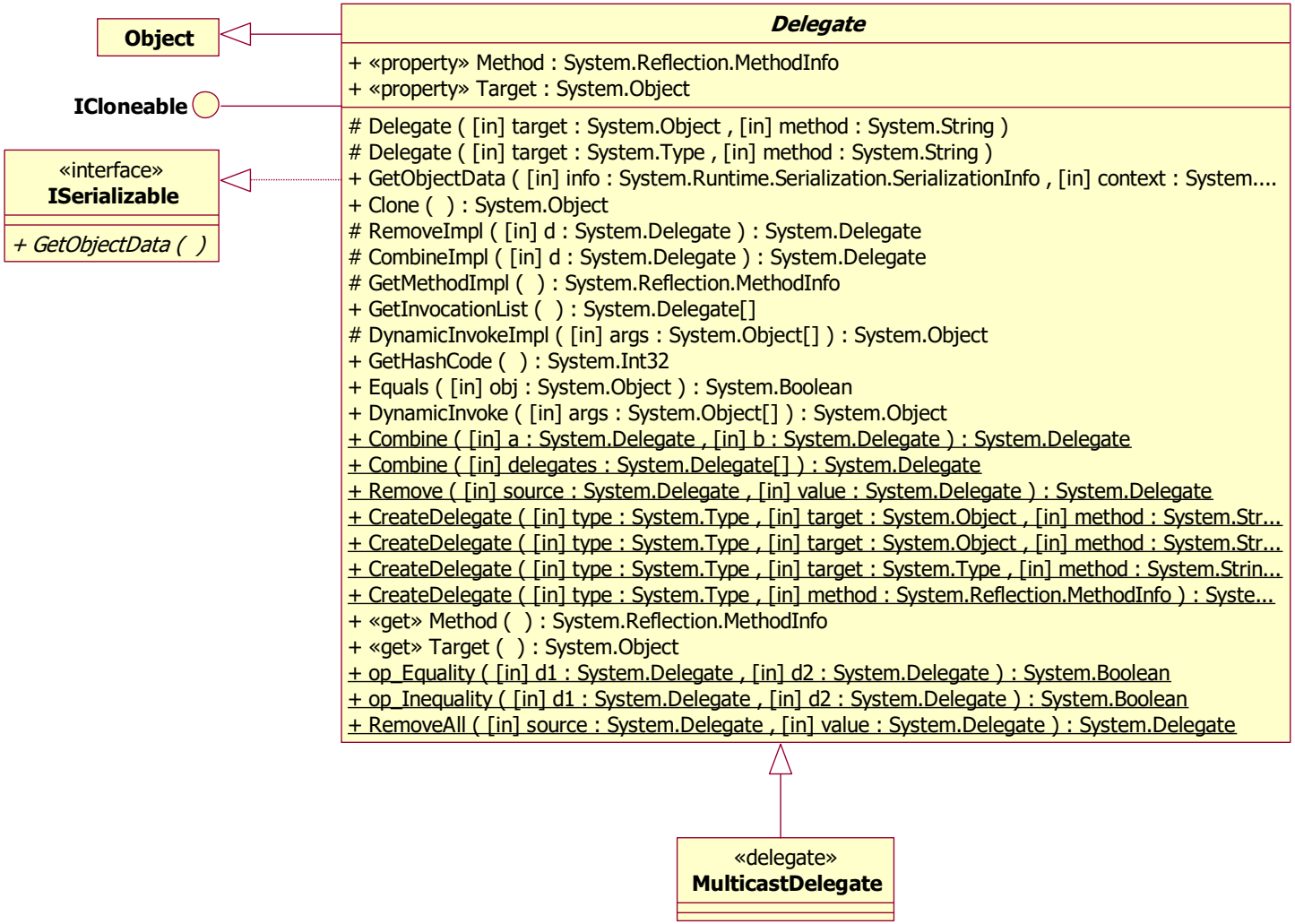


Delegati

- L’invocazione di un’istanza del delegato la cui lista di metodi contiene più elementi procede richiamando ciascuno dei metodi nella lista, **in modo sincrono**, **in ordine**
- A ogni metodo così invocato viene passato lo stesso insieme di parametri fornito all’istanza del delegato
- Se tale chiamata di delegato include **parametri di tipo riferimento**
  - ogni invocazione di metodo avverrà con un riferimento alla stessa variabile
  - le modifiche a tale variabile da parte di un metodo nella lista saranno visibili ai metodi successivi in lista
- Se la chiamata del delegato include **parametri di output** o un **valore di ritorno**
  - il loro valore finale verrà dall'invocazione dell'ultimo delegato nell'elenco

Esempio 3.3

Delegati


Object
ICloneable
«interface»
ISerializable
+ GetObjectData ( )
Delegate
+ «property» Method : System.Reflection.MethodInfo
+ «property» Target : System.Object
# Delegate ([in] target : System.Object , [in] method : System.String)
# Delegate ([in] target : System.Type , [in] method : System.String)
+ GetObjectData ([in] info : System.Runtime.Serialization.SerializationInfo , [in] context : System....
+ Clone ( ) : System.Object
# RemoveImpl ([in] d : System.Delegate) : System.Delegate
# CombineImpl ([in] d : System.Delegate) : System.Delegate
# GetMethodImpl ( ) : System.Reflection.MethodInfo
+ GetInvocationList ( ) : System.Delegate[]
# DynamicInvokeImpl ([in] args : System.Object[]) : System.Object
+ GetHashCode ( ) : System.Int32
+ Equals ([in] obj : System.Object) : System.Boolean
+ DynamicInvoke ([in] args : System.Object[]) : System.Object
+ Combine ([in] a : System.Delegate , [in] b : System.Delegate) : System.Delegate
+ Combine ([in] delegates : System.Delegate[]) : System.Delegate
+ Remove ([in] source : System.Delegate , [in] value : System.Delegate) : System.Delegate
+ CreateDelegate ([in] type : System.Type , [in] target : System.Object , [in] method : System.Str...
+ CreateDelegate ([in] type : System.Type , [in] target : System.Object , [in] method : System.Str...
+ CreateDelegate ([in] type : System.Type , [in] target : System.Type , [in] method : System.Strin...
+ CreateDelegate ([in] type : System.Type , [in] method : System.Reflection.MethodInfo) : Syste...
+ «get» Method ( ) : System.Reflection.MethodInfo
+ «get» Target ( ) : System.Object
+ op_Equality ([in] d1 : System.Delegate , [in] d2 : System.Delegate) : System.Boolean
+ op_Inequality ([in] d1 : System.Delegate , [in] d2 : System.Delegate) : System.Boolean
+ RemoveAll ([in] source : System.Delegate , [in] value : System.Delegate) : System.Delegate
«delegate»
MulticastDelegate

Delegati

- In C#, la dichiarazione di un nuovo tipo di delegato definisce automaticamente una nuova classe derivata dalla classe **`System.MulticastDelegate`**

```
System.Object
  System.Delegate
    System.MulticastDelegate
      Azione
```

- Pertanto, sulle istanze di **`Azione`** è possibile invocare i metodi definiti a livello di classi di sistema

Esempi 3.4,5,6

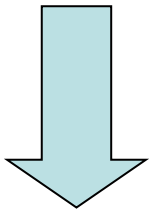
Delegati: Esempio Boss-Worker

- È necessario modellare un'interazione tra due componenti
  - un **Worker** che effettua un'attività (o lavoro)
  - un **Boss** che controlla l'attività dei suoi Worker
- Ogni Worker deve notificare al proprio Boss:
  - quando il lavoro inizia
  - quando il lavoro è in esecuzione
  - quando il lavoro finisce
- Soluzioni possibili:
  - ✓ ***class-based*** *callback relationship*
  - ✓ ***interface-based*** *callback relationship*
  - ✓ ***pattern Observer*** (lista di notifiche)
  - **4.** ***delegate-based*** *callback relationship*
  - **5.** ***event-based*** *callback relationship*

Una Relazione di Chiamata Basata su Delegati

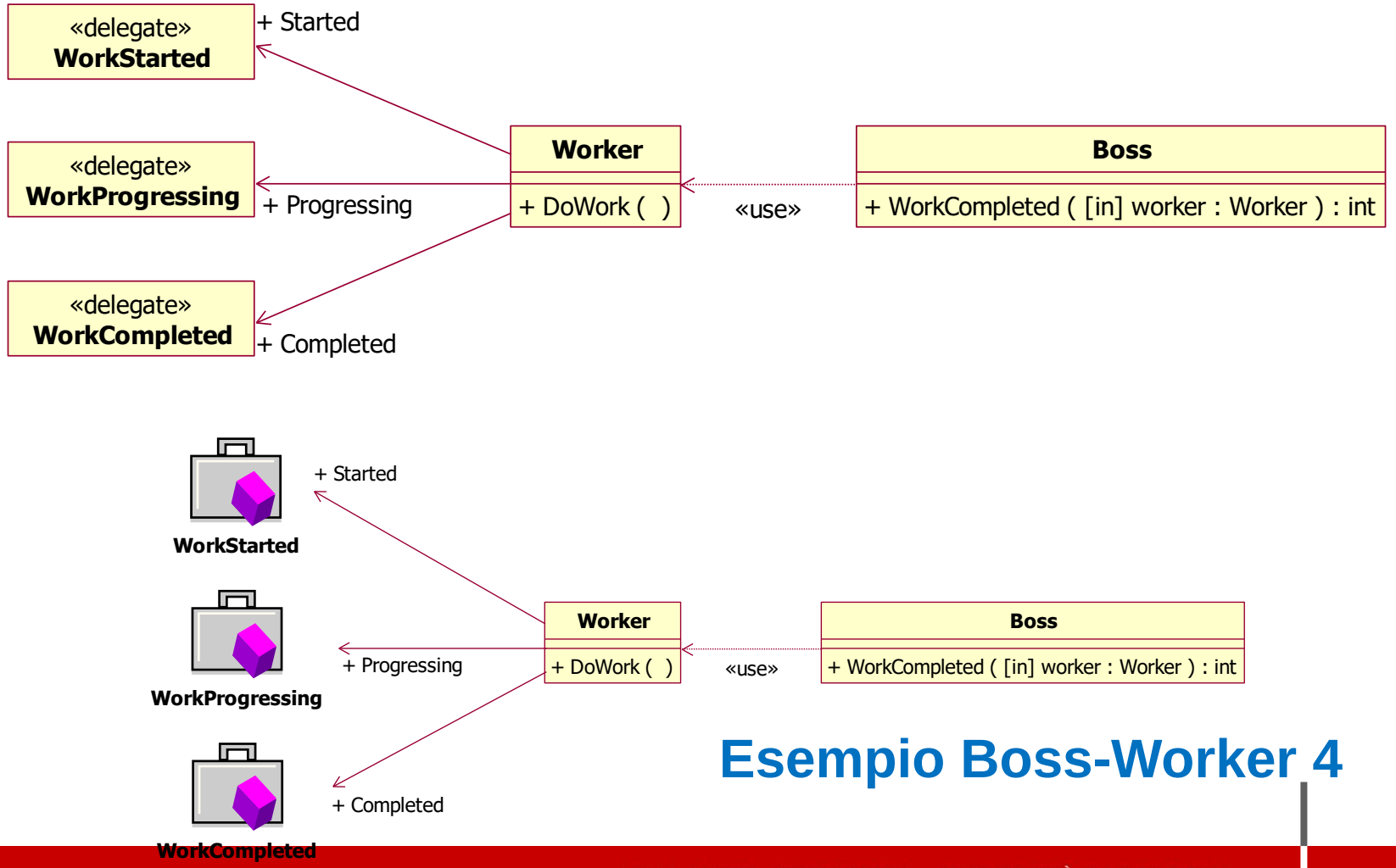
- Un delegato è un'entità *type-safe* che si pone tra 1 *caller* e 0+ *call target* e che agisce come un'interfaccia con un solo metodo

```
interface IWorkerEvents
{
   void WorkStarted(Worker worker);
   void WorkProgressing(Worker worker);
   int WorkCompleted(Worker worker);
}
```

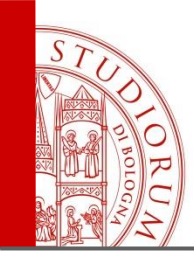


```
delegate void WorkStarted(Worker worker);
delegate void WorkProgressing(Worker worker);
delegate int WorkCompleted(Worker worker);
```

Una Relazione di Chiamata Basata su Delegati



Esempio Boss-Worker 4

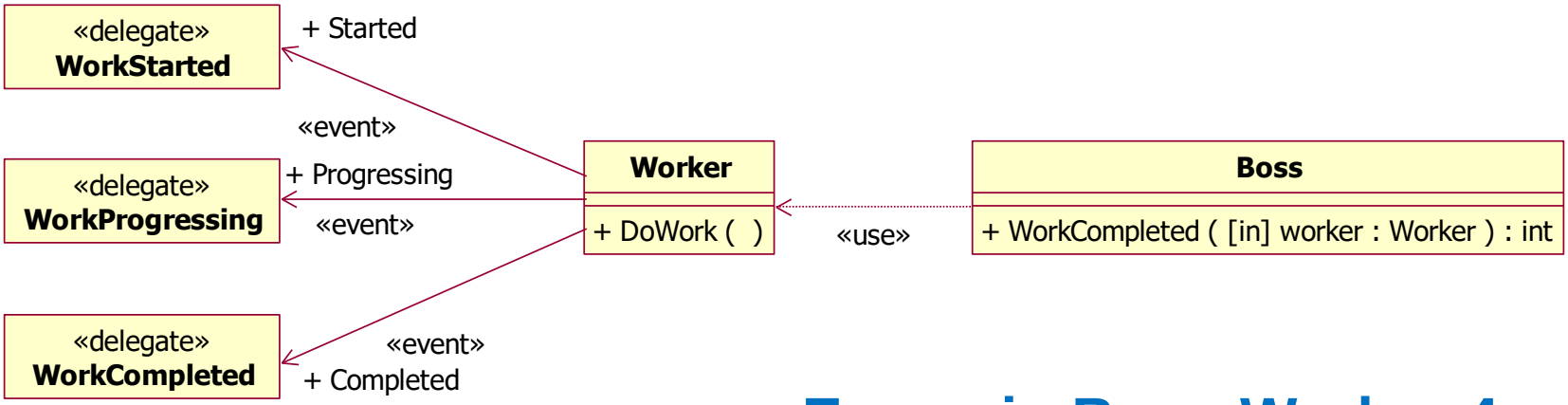


Dai Delegati agli Eventi

- L'utilizzo di campi pubblici per la registrazione fornisce un accesso eccessivo
  - I client possono sovrascrivere altri client precedentemente registrati
    `peter.Started = WorkStarted;`
  - I client possono invocare i chiamati
    `peter.Completed(peter);`
- Fornire metodi di registrazione pubblica abbinati al campo del delegato è una soluzione migliore, ma pesante se implementata manualmente
- Il modificatore **`event`** automatizza il supporto per
  - **public [un]registration** e
  - **private implementation**

Una Relazione di Chiamata Basata su Eventi

```
class Worker
{
   public event WorkStarted Started;
   public event WorkProgressing Progressing;
   public event WorkCompleted Completed;
   …
}
```



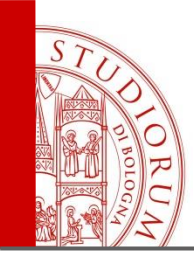
Esempio Boss-Worker 4

Personalizzare la Registrazione a Eventi

- È possibile definire gestori di registrazione a eventi
  - Uno dei vantaggi di scrivere metodi propri di registrazione è l'aumentato controllo
  - La sintassi alternativa, analoga a una proprietà, supporta gestori di registrazione definiti dall'utente
  - Consente di rendere la registrazione condizionale o diversamente personalizzata
  - Sintassi di accesso lato client non modificata
  - È necessario fornire spazio per i client registrati

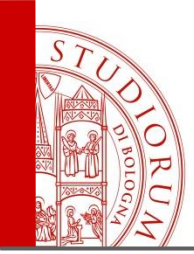
Personalizzare la Registrazione a Eventi

```
class Worker
{ …
   public event WorkProgressing Progressing
   {
     add
     {
       if(DateTime.Now.Hour < 12)
       { _progressing += value; }
       else
       { throw new InvalidOperationException
            ("Must register before noon."); }
     }
     remove
     { _progressing -= value; }
   }
   private WorkProgressing _progressing;
   …
}
```



Eventi

- **Evento:** "*Fatto o avvenimento determinante nei confronti di una situazione oggettiva o soggettiva*"
- In programmazione, un evento può essere scatenato
  - Dall'interazione con l'utente (click del mouse, …)
  - dalla logica del programma
- ***Event sender*** – l'oggetto (o la classe) che scatena (*raises* o *triggers*) l'evento (sorgente dell'evento)
- ***Event receiver*** – l'oggetto (o la classe) per il quale l'evento è determinante e che quindi desidera essere notificato quando l'evento si verifica (cliente)
- ***Event handler*** – il metodo (dell'*event receiver*) che viene eseguito all'atto della notifica



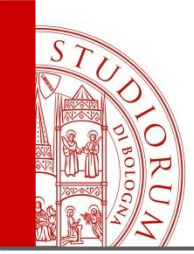
Eventi

- Quando si verifica l'evento,
  il *sender* invia un messaggio di notifica
  a tutti i *receiver*
  - in pratica, invoca gli *event handler* di tutti i *receiver*
- In genere, il *sender* NON conosce
  né i *receiver*, né gli *handler*
- Il meccanismo che viene utilizzato per collegare *sender*
  e *receiver/handler* è il **delegato**
  (che permette **invocazioni anonime**)

Dichiarazione di un Evento

Convenzione .NET

- Un evento incapsula un delegato
  ► **è necessario dichiarare un tipo di delegato prima di poter dichiarare un evento**
- Per convenzione, i delegati degli eventi in .NET hanno 2 parametri
  - la **sorgente** che ha scatenato l'evento e
  - i **dati** relativi all'evento
- Molti eventi, inclusi eventi della GUI come i click del mouse, non generano dati
- In tali situazioni, è sufficiente usare il delegato dell'evento fornito dalla libreria di classi per gli eventi senza dati, **`System.EventHandler`**
- Delegati di eventi personalizzati sono necessari solamente quando un evento genera dati



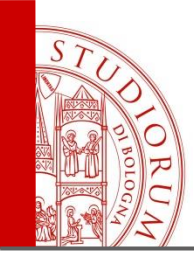
Dichiarazione di un Evento

Convenzione .NET

```
public delegate void EventHandler(
  object sender, EventArgs e);

System.Object
  System.Delegate
    System.MulticastDelegate
      System.EventHandler
```

- La classe **`System.EventArgs`** viene utilizzata quando un evento non deve passare informazioni aggiuntive ai propri gestori
- Se i gestori dell'evento hanno bisogno di informazioni aggiuntive, è necessario derivare una classe dalla classe **`EventArgs`**, aggiungere i dati necessari e utilizzare il delegate **`EventHandler<TEventArgs>`**

Esempio Boss-Worker 5



Dichiarazione di un Evento

```
public event EventHandler Changed;
```

- In pratica, **`Changed`** è un delegato, ma la *keyword* **`event`** ne limita
  - la visibilità e
  - le possibilità di utilizzo

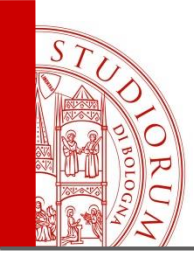
- Una volta dichiarato, l'evento può essere trattato come un delegato di tipo speciale
- In particolare, può:
  - essere **`null`** se nessun cliente si è registrato
  - essere associato a uno o più metodi da invocare

Invocazione di un Evento

- Per scatenare un evento è opportuno definire un metodo protetto virtuale **OnNomeEvento** e invocare sempre quello

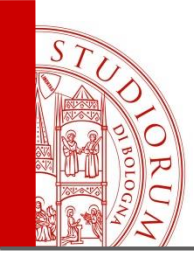
```
public event EventHandler Changed;

protected virtual void OnChanged()
{
  if(Changed != null)
    Changed(this, EventArgs.Empty);
}
…
OnChanged();
…
```

- **Limitazione rispetto ai delegati**
  L'invocazione dell'evento può avvenire solo all'interno della classe nella quale l'evento è stato dichiarato (benché l'evento sia stato dichiarato **public**)



Utilizzo di un Evento

- Al di fuori della classe in cui l'evento è stato dichiarato, un evento viene visto come un **delegato con accessi molto limitati**
- Le sole operazioni effettuabili dal cliente sono:
  - **agganciarsi a un evento**: aggiungere un nuovo delegato all'evento mediante l'operatore **+=**
  - **sganciarsi da un evento**: rimuovere un delegato dall'evento mediante l'operatore **−=**



Agganciarsi a un Evento

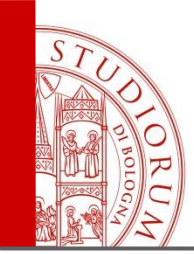
Per iniziare a ricevere le notifiche di un evento, il cliente deve:

- **Definire il metodo** (*event handler*) **che dovrà essere invocato** all'atto della notifica dell'evento (con la stessa *signature* dell'evento):

```
void ListChanged(object sender, EventArgs e)
{ … }
```

- **Creare un delegato** dello stesso tipo dell'evento, farlo riferire al metodo e **aggiungerlo** alla lista dei delegati associati **all'evento**:

```
List.Changed += new EventHandler(ListChanged);

List.Changed += ListChanged;  //  C# 2.0
```



Sganciarsi da un Evento

Per smettere di ricevere le notifiche di un evento, il cliente deve:

- **Rimuovere il delegato** dalla lista dei delegati associati all'evento:

```
List.Changed -= new EventHandler(ListChanged);
List.Changed -= ListChanged;  //  C# 2.0
```

Eventi

- Poiché += e −= sono gli unici operatori permessi su un evento al di fuori del tipo che dichiara l'evento, il codice esterno al tipo
  - può aggiungere e rimuovere handler per un evento, ma
  - non può in nessun altro modo ottenere o modificare la lista di handler sottostante

- Gli eventi forniscono agli oggetti un modo utile per segnalare modifiche allo stato a clienti di tali oggetti
- Gli eventi sono un componente fondamentale **per la creazione di classi che possano essere riutilizzate in un gran numero di programmi differenti**

Esempio MVC ► MVP